

# Chapter 5

## Write Perl Client Applications

Juniper Networks provides a Perl module, called JUNOS, to help you more quickly and easily develop custom Perl scripts for configuring and monitoring routers. The module implements an object, called JUNOS::Device, that client applications can use to communicate with the JUNOScript server on a router. Accompanying the JUNOS module are several sample Perl scripts, which illustrate how to use the module in scripts that perform various functions.

This chapter discusses the following topics:

- Overview of the JUNOS Module and Sample Scripts on page 81
- Download the JUNOS Module and Sample Scripts on page 82
- Tutorial: Writing Perl Client Applications on page 83
- Summary of Mappings Between Perl Queries and JUNOScript Tag Elements on page 106

### Overview of the JUNOS Module and Sample Scripts

The JUNOScript Perl distribution uses the same directory structure for Perl modules as the Comprehensive Perl Archive Network (<http://www.cpan.org>). This includes a lib directory for the JUNOS module and its supporting files, and an examples directory for the sample scripts.

The JUNOS module implements an object (JUNOS::Device) that client applications can use to communicate with a JUNOScript server. All of the sample scripts use the object.

The sample scripts illustrate how to perform the following functions:

`diagnose_bgp.pl`—Illustrates how to write scripts to monitor router status and diagnose problems. The sample script extracts and displays information about a router's unestablished Border Gateway Protocol (BGP) peers from the full set of BGP configuration data. The script is provided in the examples/diagnose\_bgp directory in the JUNOScript Perl distribution.

`get_chassis_inventory.pl`—Illustrates how to use one of the predefined Perl JUNOScript queries to request information from a router. The sample script invokes the `get_chassis_inventory` query with the detail option to request the same information as the JUNOScript `<get-chassis-inventory><detail/></get-chassis-inventory>` tag sequence and the JUNOS command-line interface (CLI) `show chassis hardware detail` command. The script is provided in the examples/get\_chassis\_inventory directory in the JUNOScript Perl distribution. For a list of all Perl queries available in this release of JUNOScript, see Table 6 on page 107.

- `load_configuration.pl`—Illustrates how to change router configuration by loading a file that contains configuration data formatted with JUNOScript tag elements. The distribution includes two sample configuration files, `set_login_user_foo.xml` and `set_login_class_bar.xml`; however, you can specify another JUNOScript configuration file on the command line. The script is provided in the `examples/load_configuration` directory in the JUNOScript Perl distribution.

The following sample scripts are used together to illustrate how to store and retrieve JUNOScript (or any Extensible Markup Language [XML]) data in a relational database. While these scripts create and manipulate MySQL tables, the data manipulation techniques that they illustrate apply to any relational database. The scripts are provided in the `examples/RDB` directory in the JUNOScript Perl distribution:

- `get_config.pl`—Illustrates how to retrieve router configuration information.
- `make_tables.pl`—Generates a set of Structured Query Language (SQL) statements for creating relational database tables.
- `pop_tables.pl`—Populates existing relational database tables with data extracted from a specified XML file.
- `unpop_tables.pl`—Transforms data stored in a relational database table into XML and writes it to a file.

For instructions on running the scripts, see the `README` or `README.html` file included in the JUNOScript Perl distribution.

## Download the JUNOS Module and Sample Scripts

To download, uncompress, and unpack the compressed tar-format file that contains the JUNOS module and sample scripts, perform the following steps:

1. Access the Juniper Networks Customer Support Center Web page at <http://www.juniper.net/support>.
2. Click on the link labeled JUNOScript API Software.
3. On the JUNOScript API Software Download page, click on the link for the appropriate JUNOS Internet software release.
4. To download the JUNOScript API Perl client package and the prerequisites package, click on the links for the packages that support the appropriate access protocols. Customers in the United States and Canada can download the package that supports all access protocols (the domestic package), or the package that supports clear-text and telnet only (the export package). Customers in other countries can download the package that supports clear-text and telnet only.



It is assumed that the machine on which you store and run the Perl client software is a regular computer instead of a Juniper Networks router.

**Note**

5. Optionally, download the package containing document type definitions (DTDs).

6. Change to the directory where you want to create a subdirectory that contains the JUNOS Perl module and sample scripts:

```
% cd perl-parent-directory
```

7. Issue the following command to uncompress and unpack the package downloaded in Step 4:

On FreeBSD and Linux systems:

```
% tar zxf junoscript-perl-release-type.tar.gz
```

On Solaris systems:

```
% gzip -dc junoscript-perl-release-type.tar.gz | tar xf
```

where *release* is the JUNOS release code (such as 5.6R1.1) and *type* is domestic or export. The command creates a directory called *junoscript-perl-release-type* and writes the contents of the tar file to it.

8. See the *junoscript-perl-release-type/README* file for instructions on unpacking and installing the Perl prerequisite modules, creating a Makefile, and installing and testing the JUNOS module.

## Tutorial: Writing Perl Client Applications

This tutorial explains how to write a Perl client application that requests operational or configuration information from the JUNOScript server or loads configuration information onto a router. The following sections use the sample scripts included in the JUNOScript Perl distribution as examples:

[Import Perl Modules and Declare Constants on page 84](#)

[Connect to the JUNOScript Server on page 84](#)

[Submit a Request to the JUNOScript Server on page 91](#)

[Parse and Format the Response from the JUNOScript Server on page 100](#)

[Close the Connection to the JUNOScript Server on page 106](#)

## Import Perl Modules and Declare Constants

Include the following statements at the start of the application. The first statement imports the functions provided by the JUNOS::Device object, which the application uses to connect to the JUNOScript server on a router. The second statement provides error checking and enforces Perl coding practices such as declaration of variables before use.

```
use JUNOS::Device;
use strict;
```

Include other statements to import other Perl modules as appropriate for your application. For example, several of the sample scripts import the following standard Perl modules, which include functions that handle input from the command line:

Getopt::Std—Includes functions for reading in keyed options from the command line.

Term::ReadKey—Includes functions for controlling terminal modes, for example suppressing onscreen echo of a typed string such as a password.

File::Basename—Includes functions for processing filenames.

If the application uses constants, declare their values at this point. For example, the sample diagnose\_bgp.pl script includes the following statements to declare constants for formatting output:

```
use constant OUTPUT_FORMAT => "%-20s%-8s%-8s%-11s%-14s%\n";
use constant OUTPUT_TITLE => "\n===== BGP PROBLEM SUMMARY
=====\\n\\n";
use constant OUTPUT_ENDING =>
"\n=====
==\\n\\n";
```

The load\_configuration.pl script includes the following statements to declare constants for reporting return codes and the status of the configuration database:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

## Connect to the JUNOScript Server

The following sections explain how to use the JUNOS::Device object to connect to the JUNOScript server on a router:

[Satisfy Protocol Prerequisites on page 85](#)

[Group Requests on page 85](#)

[Obtain and Record Parameters Required by the JUNOS::Device Object on page 85](#)

[Obtain Application-Specific Parameters on page 88](#)

[Convert Disallowed Characters on page 89](#)

[Establish the Connection on page 91](#)

## Satisfy Protocol Prerequisites

The JUNOScript server supports several access protocols, listed in “Supported Access Protocols” on page 14. For each connection to the JUNOScript server on a router, the application must specify the protocol it is using. Using secure shell (ssh) or Secure Sockets Layer (SSL) is recommended because they provide greater security by encrypting all information before transmission across the network.

Before your application can run, you must satisfy the prerequisites for the protocol it uses. For some protocols this involves activating configuration statements on the router, creating encryption keys, or installing additional software on the router or the machine where the application runs. For instructions, see “Prerequisites for Establishing a Connection” on page 15.

## Group Requests

Establishing a connection to the JUNOScript server on a router is one of the more time- and resource-intensive functions performed by an application. If the application sends multiple requests to a router, it makes sense to send all of them within the context of one connection. If your application sends the same requests to multiple routers, you can structure the script to iterate through either the set of routers or the set of requests. Keep in mind, however, that your application can effectively send only one request to one JUNOScript server at a time. This is because the JUNOS::Device object does not return control to the application until it receives the closing </rpc-reply> tag that represents the end of the JUNOScript server’s response to the current request.

## Obtain and Record Parameters Required by the JUNOS::Device Object

The JUNOS::Device object takes the following required parameters, specified as keys in a Perl hash:

The access protocol to use when communicating with the JUNOScript server (key name: access). For a list of the acceptable values, see “Supported Access Protocols” on page 14. Before the application runs, satisfy the protocol-specific prerequisites described in “Prerequisites for Establishing a Connection” on page 15.

The name of the router to which to connect (key name: hostname). For best results, specify either a fully-qualified hostname or an IP address.

The username of the JUNOS login account under which to establish the JUNOScript connection and issue requests (key name: login). The account must already exist on the specified router and have the JUNOS permission bits necessary for making the requests invoked by the application.

The password for the JUNOS login account (key name: password).

The sample scripts record the parameters in a Perl hash called %deviceinfo, declared as follows:

```
my %deviceinfo = (
    access => $access,
    login => $login,
    password => $password,
    hostname => $hostname,
);
```

- The sample scripts obtain the parameters from options entered on the command line by a user. Your application can also obtain values for the parameters from a file or database, or you can hardcode one or more of the parameters into the application code if they are constant.

*Example: Collect Parameters Interactively*

Each sample script obtains the parameters required by the JUNOS::Device object from command-line options provided by the user who invokes the script. The script records the options in a Perl hash called %opt, using the getopts function defined in the Getopt::Std Perl module to read the options from the command line. (Scripts used in production environments probably do not obtain parameters interactively, so this section is important mostly for understanding the sample scripts.)

In the following example from the `get_chassis_inventory.pl` script, the first parameter to the `getopts` function defines the acceptable options, which vary depending on the application. A colon after the option letter indicates that it takes an argument. The second parameter, `\%opt`, specifies that the values are recorded in the `%opt` hash. If the user does not provide at least one option, provides an invalid option, or provides the `-h` option, the script invokes the `output_usage` subroutine, which prints a usage message to the screen:

```
my %opt;
getopts('I:p:dm:x:o:h', \%opt) || output_usage();
output_usage() if $opt{h};
```

The following code defines the `output_usage` subroutine for the `get_chassis_inventory.pl` script. The contents of the `my $usage` definition and the `Where` and `Options` sections are specific to the script, and differ for each application.

```
sub output_usage
{
    my $usage = "Usage: $0 [options] <target>
```

Where:

**<target>** The hostname of the target router

#### Options:

- l <login> A login name accepted by the target router.
  - p <password> The password for the login name.
  - m <access> Access method. It can be clear-text, ssl, ssh or telnet. Default: telnet.
  - x <format> The name of the XSL file to display the response.  
Default: xsl/chassis\_inventory\_csv.xsl
  - o <filename> File to which to write output, instead of standard output.
  - d Turn on debugging.\n\n";

```
    die $usage;  
}
```

The `get_chassis_inventory.pl` script includes the following code to obtain values from the command line for the four parameters required by the `JUNOS::Device` object. A detailed discussion of the various functional units follows the complete code sample.

```
my $hostname = shift || output_usage();

my $access = $opt{m} || "telnet";
use constant VALID_ACCESES => "telnet|ssh|clear-text|ssl";
output_usage() unless (VALID_ACCESES =~ /$access/);

my $login = "";
if ($opt{l}) {
    $login = $opt{l};
} else {
    print "login: ";
    $login = ReadLine 0;
    chomp $login;
}

my $password = "";
if ($opt{p}) {
    $password = $opt{p};
} else {
    print "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print "\n";
}
```

In the first line of the preceding code sample, the script uses the Perl `shift` function to read the hostname from the end of the command line. If the hostname is missing, the script invokes the `output_usage` subroutine to print the usage message, which specifies that a hostname is required:

```
my $hostname = shift || output_usage();
```

The script next determines which access protocol to use, setting the `$access` variable to the value of the `-m` command-line option or to the value `telnet` if the `-m` option is not provided. If the specified value does not match one of the four values defined by the `VALID_ACCESES` constant, the script invokes the `output_usage` subroutine:

```
my $access = $opt{m} || "telnet";
use constant VALID_ACCESES => "telnet|ssh|clear-text|ssl";
output_usage() unless ($access =~ /VALID_ACCESES/);
```

The script then determines the JUNOS login account name, setting the `$login` variable to the value of the `-l` command-line option. If the option is not provided, the script prompts for it and uses the `ReadLine` function (defined in the standard Perl `Term::ReadKey` module) to read it from the command line:

```
my $login = "";
if ($opt{l}) {
    $login = $opt{l};
} else {
    print "login: ";
    $login = ReadLine 0;
    chomp $login;
}
```

- The script finally determines the password for the JUNOS account, setting the \$password variable to the value of the -p command-line option. If the option is not provided, the script prompts for it. It uses the ReadMode function (defined in the standard Perl Term::ReadKey module) twice: first to prevent the password from echoing visibly on the screen and then to return the shell to normal (echo) mode after it reads the password:

```
my $password = "";
if ($opt{p}) {
    $password = $opt{p};
} else {
    print "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print "\n";
}
```

## ***Obtain Application-Specific Parameters***

In addition to the parameters required by the JUNOS::Device object, applications might need to define other parameters, such as the name of the file to which to write the data returned by the JUNOScript server in response to a request, or the name of the Extensible Stylesheet Transformation Language (XSLT) file to use for transforming the data.

As with the parameters required by the JUNOS::Device object, your application can hardcode the values in the application code, obtain them from a file, or obtain them interactively. The sample scripts obtain values for these parameters from command-line options in the same manner as they obtain the parameters required by the JUNOS::Device object (discussed in “Obtain and Record Parameters Required by the JUNOS::Device Object” on page 85). Several examples follow.

The following line enables a debugging trace if the user includes the -d command-line option. It invokes the JUNOS::Trace::init routine defined in the JUNOS::Trace module, which is already imported with the JUNOS::Device object.

```
JUNOS::Trace::init(1) if $opt{d};
```

The following line sets the `$outputfile` variable to the value specified by the `-o` command-line option. It names the local file to which the JUNOScript server's response is written. If the `-o` option is not provided, the variable is set to the empty string.

```
my $outputfile = $opt{o} || "out";
```

The following code from the `diagnose_bgp.pl` script defines which XSLT file to use to transform the JUNOScript server's response. The first line sets the `$xslfile` variable to the value specified by the `-x` command-line option. If the option is not provided, the script uses the `text.xsl` file supplied with the script, which transforms the data to ASCII text. The `if` statement verifies that the specified XSLT file exists; the script terminates if it does not.

```
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
}
```

The following code from the load\_configuration.pl script defines whether to merge, replace, or overwrite the new configuration data into the configuration database (for more information about these operations, see “Change the Candidate Configuration” on page 61). The first two lines set the \$load\_action variable to the value of the -a command-line option, or to the default value merge if the option is not provided. If the specified value does not match one of the four defined in the third line, the script invokes the output\_usage subroutine.

```
# The default action for load_configuration is 'merge'
my $load_action = "merge";
$load_action = $opt{a} if $opt{a};
use constant VALID_ACTIONS => "merge|replace|override";
output_usage() unless ($load_action =~ /VALID_ACTIONS/);
```

### **Convert Disallowed Characters**

Scripts that handle configuration data usually accept and output the data either as JUNOScript tag elements or as ASCII-formatted statements like those used in the JUNOS CLI. As described in “Predefined Entity References” on page 12, certain characters cannot appear in their regular form in an XML document. These characters include the apostrophe ( ‘ ), the ampersand ( & ), the greater-than ( > ) and less-than ( < ) symbols, and the quotation mark ( “ ” ). Because these characters might appear in ASCII-formatted configuration statements, the script must convert the characters to the corresponding predefined entity references.

The load\_configuration.pl script uses the get\_escaped\_text subroutine to substitute predefined entity references for disallowed characters (the get\_configuration.pl script includes similar code). The script first defines the mappings between the disallowed characters and predefined entity references, and sets the variable \$char\_class to a regular expression that contains all of the entity references, as follows:

```
my %escape_symbols = (
    qq(") => '"',
    qq(>) => '>',
    qq(<) => '<',
    qq(') => ''',
    qq(&) => '&#';
);

my $char_class = join ("|", map { "$_" } keys %escape_symbols);
```

- The following code defines the `get_escaped_text` subroutine for the `load_configuration.pl` script. A detailed discussion of the subsections in the routine follows the complete code sample.

```

sub get_escaped_text
{
    my $input_file = shift;
    my $input_string = "";

    open(FH, $input_file) or return undef;

    while(<FH>) {
        my $line = $_;
        $line =~ s/<configuration-text>//g;
        $line =~ s/<\>/configuration-text>/g;
        $line =~ s/($char_class)/$escape_symbols{$1}/ge;
        $input_string .= $line;
    }

    return "<configuration-text>$input_string</configuration-text>";
}

```

The first subsection of the preceding code sample reads in a file containing ASCII-formatted configuration statements:

```

sub get_escaped_text
{
    my $input_file = shift;
    my $input_string = "";

    open(FH, $input_file) or return undef;

```

In the next subsection, the subroutine temporarily discards the lines that contain the opening `<get-configuration>` and closing `</get-configuration>` tags, then replaces the disallowed characters on each remaining line with predefined entity references and appends the line to the `$input_string` variable:

```

while(<FH>) {
    my $line = $_;
    $line =~ s/<configuration-text>//g;
    $line =~ s/<\>/configuration-text>/g;
    $line =~ s/($char_class)/$escape_symbols{$1}/ge;
    $input_string .= $line;
}

```

The subroutine concludes by replacing the opening `<get-configuration>` and closing `</get-configuration>` tags, and returning the converted set of statements:

```

return "<configuration-text>$input_string</configuration-text>";
}

```

## **Establish the Connection**

After obtaining values for the parameters required for the JUNOS::Device object (see “Obtain and Record Parameters Required by the JUNOS::Device Object” on page 85), each sample script records them in the %deviceinfo hash:

```
my %deviceinfo = (
    access => $access,
    login => $login,
    password => $password,
    hostname => $hostname,
);
```

The script then invokes the JUNOScript-specific new subroutine to create a JUNOS::Device object and establish a connection to the specified router. If the connection attempt fails (as tested by the ref operator), the script exits.

```
my $jnx = new JUNOS::Device(%deviceinfo);
unless ( ref $jnx ) {
    die "ERROR: $deviceinfo(hostname): failed to connect.\n";
```

## **Submit a Request to the JUNOScript Server**

After establishing a connection to a JUNOScript server (see “Establish the Connection” on page 91), your application can submit one or more requests by invoking the Perl methods that are supported in the version of the JUNOScript API used by the application:

Each version of JUNOS software supports a set of methods that correspond to JUNOS CLI operational mode commands (later releases generally support more methods). For a list of the operational methods supported in the current version, see “Summary of Mappings Between Perl Queries and JUNOScript Tag Elements” on page 106 and the files stored in the lib/JUNOS/release directory of the JUNOScript Perl distribution (*release* is the version code such as 5.6R1 for the initial release of JUNOS 5.6). The files have names in the format *package*\_methods.pl, where *package* is a JUNOS software package.

The set of methods that correspond to operations on JUNOS configuration objects is defined in the file lib/JUNOS/Methods.pm in the JUNOScript Perl distribution. For more information about configuration operations, see “Change the Candidate Configuration” on page 61 and the chapter about session control tags in the *JUNOScript API Reference*.

See the following sections for more information:

[Provide Method Options or Attributes on page 92](#)

[Submit a Request on page 94](#)

[Example: Get an Inventory of Hardware Components on page 95](#)

[Example: Load Configuration Statements on page 96](#)

- **Provide Method Options or Attributes**

Many Perl methods have one or more options or attributes. The following list describes the notation used to define a method's options in the lib/JUNOS/Methods.pm and lib/JUNOS/release/package\_methods.pl files, and the notation that an application uses when invoking the method:

A method without options is defined as \$NO\_ARGS, as in the following entry for the get\_system\_uptime\_information method:

```
## Method : <get-system-uptime-information>
## Returns: <system-uptime-information>
## Command: "show system uptime"
get_system_uptime_information => $NO_ARGS,
```

To invoke a method without options, follow the method name with an empty set of parentheses as in the following example:

```
$jnx->get_system_uptime_information();
```

A fixed-form option is defined as type \$TOGGLE. In the following example, the get\_software\_information method takes two fixed-form options, brief and detail:

```
## Method : <get-software-information>
## Returns: <software-information>
## Command: "show version"
get_software_information => {
    brief => $TOGGLE,
    detail => $TOGGLE,
},
```

To include a fixed-form option when invoking a method, set it to the value 1 (one) as in the following example:

```
$jnx->get_software_information(brief => 1);
```

An option with a variable value is defined as type \$STRING. In the following example, the get\_cos\_drop\_profile\_information method takes the profile\_name argument:

```
## Method : <get-cos-drop-profile-information>
## Returns: <cos-drop-profile-information>
## Command: "show class-of-service drop-profile"
get_cos_drop_profile_information => {
    profile_name => $STRING,
};
```

To include a variable value when invoking a method, enclose the value in single quotes as in the following example:

```
$jnx->get_cos_drop_profile_information(profile_name => 'user-drop-profile');
```

An attribute is defined as type \$ATTRIBUTE. In the following example, the lock\_configuration method takes the rollback attribute:

```
lock_configuration => {
    rollback => $ATTRIBUTE
},
```

To include a numerical attribute value when invoking a method, set it to the appropriate value. The following example rolls the candidate configuration back to the previous configuration that has an index of 2:

```
$jnx->load_configuration(rollback => 2);
```

To include a string attribute value when invoking a method, enclose the value in single quotes as in the following example:

```
$jnx->get_configuration(format => 'text');
```

A set of configuration statements or corresponding tag elements is defined as type \$DOM. In the following example, the get\_configuration method takes a set of configuration statements (along with two attributes):

```
get_configuration => {
    configuration => $DOM,
    format => $ATTRIBUTE,
    database => $ATTRIBUTE,
},
```

To include a set of configuration statements when invoking a method, provide a parsed set of statements or tags. The following example refers to a set of JUNOScript configuration tags in the config-input.xml file. For further discussion, see “Example: Load Configuration Statements” on page 96.

```
my $parser = new XML::DOM::Parser;
$jnx->load_configuration(
    format => 'xml',
    action => 'merge',
    configuration => $parser->parsefile(config-input.xml)
);
```

- A method can have a combination of fixed-form options, options with variable values, attributes, and a set of configuration statements. For example, the `get_route_forwarding_table` method has four fixed-form options and five options with variable values:

```
## Method : <get-forwarding-table-information>
## Returns: <forwarding-table-information>
## Command: "show route forwarding-table"
get_forwarding_table_information => {
    detail => $TOGGLE,
    extensive => $TOGGLE,
    multicast => $TOGGLE,
    family => $STRING,
    vpn => $STRING,
    summary => $TOGGLE,
    matching => $STRING,
    destination => $STRING,
    label => $STRING,
},
} ,
```

## ***Submit a Request***

The following is the recommended way to send a request to the JUNOScript server. It assumes that the \$jnx variable was previously defined to be a JUNOS::Device object, as discussed in “Establish the Connection” on page 91.

The following code sends a request to the JUNOScript server and handles error conditions. A detailed discussion of the functional subsections follows the complete code sample.

```

my %arguments = ();
%arguments = (argument1 => value1,
    argument2 => value2, ...);
    argument3 => value3,
...);

my $res = $jnx->method(%args);

unless ( ref $res ) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Could not send request to $hostname\n";
}

my $err = $res->getFirstError();
if ($err) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}

```

The first subsection of the preceding code sample creates a hash called %arguments to define values for a method's options or attributes. For each argument, the application uses the notation described in "Provide Method Options or Attributes" on page 92.

```
my %arguments = ();  
%arguments = (argument1 => value1,  
             argument2 => value2, ...);  
             argument3 => value3,  
             ...);
```

The application then invokes the method, defining the \$res variable to point to the JUNOS::Response object that the JUNOScript server returns in response to the request (the object is defined in the lib/JUNOS/Response.pm file in the JUNOScript Perl distribution):

```
my $res = $jnx->method(%args);
```

If the attempt to send the request failed, the application prints an error message and closes the connection:

```
unless ( ref $res ) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Could not send request to $hostname\n";
}
```

If there was an error in the JUNOScript server's response, the application prints an error message and closes the connection. The getFirstError function is defined in the JUNOS::Response module (lib/JUNOS/Response.pm) in the JUNOScript Perl distribution.

```
my $err = $res->getFirstError();
if ($err) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}
```

### **Example: Get an Inventory of Hardware Components**

The get\_chassis\_inventory.pl script retrieves and displays a detailed inventory of the hardware components installed in a router. It is equivalent to issuing the show chassis hardware detail command.

After establishing a connection to the JUNOScript server, the script defines get\_chassis\_inventory as the request to send and includes the detail argument:

```
my $query = "get_chassis_inventory";
my %queryargs = ( detail => 1 );
```

The script sends the query and assigns the results to the \$res variable. It performs two tests on the results, and prints an error message if it cannot send the request or if errors occurred when executing it. If no errors occurred, the script uses XSLT to transform the results. For more information, see “Parse and Format an Operational Response” on page 101.

```
my $res = $jnx->$query( %queryargs );
unless ( ref $res ) {
    die "ERROR: $deviceinfo(hostname): failed to execute command $query.\n";
}
my $err = $res->getFirstError();
if ($err) {
    print STDERR "ERROR: $deviceinfo{'hostname'} - ", $err->{message}, "\n";
} else {
    ... code to process results with XSLT ...
}
```

- **Example: Load Configuration Statements**
- 

The load\_configuration.pl script loads configuration statements onto a router. It uses the basic structure for sending requests described in “Submit a Request” on page 94, but also defines a graceful\_shutdown subroutine that handles errors in a slightly more elaborate manner than that described in “Submit a Request” on page 94. The following sections describe the different functions that the script performs:

- Handle Error Conditions on page 96
- Lock the Configuration on page 97
- Read In and Parse the Configuration Data on page 98
- Load the Configuration Data on page 99
- Commit the Configuration on page 100

- *Handle Err or Conditions*
- 

The graceful\_shutdown subroutine in the load\_configuration.pl script handles errors in a slightly more elaborate manner than the generic structure described in “Submit a Request” on page 94. It employs the following additional constants:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

The first two if statements in the subroutine refer to the STATE\_CONFIG\_LOADED and STATE\_LOCKED conditions, which apply specifically to loading a configuration in the load\_configuration.pl script. The if statement for STATE\_CONNECTED is similar to the error checking described in “Submit a Request” on page 94. The eval statement used in each case ensures that any errors that occur during execution of the enclosed function call are trapped so that failure of the function call does not cause the script to exit.

```
sub graceful_shutdown
{
    my ($jnx, $req, $state, $success) = @_;
    if ($state >= STATE_CONFIG_LOADED) {
        print "Rolling back configuration ...\\n";
        eval {
            $jnx->load_configuration(rollback => 0);
        };
    }
    if ($state >= STATE_LOCKED) {
        print "Unlocking configuration database ...\\n";
        eval {
            $jnx->unlock_configuration();
        };
    }
    if ($state >= STATE_CONNECTED) {
        print "Disconnecting from the router ...\\n";
        eval {
            $jnx->request_end_session();
            $jnx->disconnect();
        };
    }
    if ($success) {
        die "REQUEST $req SUCCEEDED\\n";
    } else {
        die "REQUEST $req FAILED\\n";
    };
}
```

### *Lock the Configuration*

The main section of the load\_configuration.pl script begins by establishing a connection to a JUNOScript server, as described in “Establish the Connection” on page 91. It then invokes the lock\_configuration method to lock the configuration database. In case of error, the script invokes the graceful\_shutdown subroutine described in “Handle Error Conditions” on page 96.

```
print "Locking configuration database ...\\n";
my $res = $jnx->lock_configuration();
my $err = $res->getFirstError();
if ($err) {
    print "ERROR: $deviceinfo(hostname): failed to lock configuration. Reason:
$err->(message).\\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONNECTED, REPORT_FAILURE);
}
```

- *Read In and Parse the Configuration Data*

In the following code sample, the load\_configuration.pl script then reads in and parses a file that contains JUNOScript configuration tag elements or ASCII-formatted statements. The name of the file was previously obtained from the command line and assigned to the \$xmlfile variable. A detailed discussion of the functional subsections follows the complete code sample.

```

print "Loading configuration from $xmlfile ...\\n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

my $parser = new XML::DOM::Parser;
...
my $doc;
if ($opt{t}) {
    my $xmlstring = get_escaped_text($xmlfile);
    $doc = $parser->parsestring($xmlstring) if $xmlstring;

} else {
    $doc = $parser->parsefile($xmlfile);
}

unless ( ref $doc ) {
    print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is well-formed\\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

```

The first subsection of the preceding code sample verifies the existence of the file containing configuration data. The name of the file was previously obtained from the command line and assigned to the \$xmlfile variable. If the file does not exist, the script invokes the graceful\_shutdown subroutine:

```

print "Loading configuration from $xmlfile ...\\n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

```

If the -t command-line option was included when the load\_configuration.pl script was invoked, the file referenced by the \$xmlfile variable should contain ASCII-formatted configuration statements like those returned by the CLI configuration-mode show command. The script invokes the get\_escaped\_text subroutine described in “Convert Disallowed Characters” on page 89, assigning the result to the \$xmlstring variable. The script invokes the parsestring function to transform the data in the file into the proper format for loading into the configuration hierarchy, and assigns the result to the \$doc variable. The parsestring function is defined in the XML::DOM::Parser module, and the first line in the following sample code instantiates the module as an object, setting the \$parser variable to refer to it:

```

my $parser = new XML::DOM::Parser;
...
my $doc;
if ($opt{t}) {
    my $xmlstring = get_escaped_text($xmlfile);
    $doc = $parser->parsestring($xmlstring) if $xmlstring;
}

```

If the file contains JUNOScript configuration tags instead, the script invokes the `parsefile` function (also defined in the `XML::DOM::Parser` module) on the file:

```
    } else {
        $doc = $parser->parsefile($xmlfile);
    }
```

If the parser cannot transform the file, the script invokes the `graceful_shutdown` subroutine described in “Handle Error Conditions” on page 96:

```
unless ( ref $doc ) {
    print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is well-formed\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

### *Load the Configuration Data*

The script now invokes the `load_configuration` method to load the configuration onto the router. It places the statement inside an `eval` block to ensure that the `graceful_shutdown` subroutine is invoked if the response from the JUNOScript server has errors.

```
eval {
$res = $jnx->load_configuration(
    format => $config_format,
    action => $load_action,
    configuration => $doc);
};

if ($@) {
    print "ERROR: Failed to load the configuration from $xmlfile. Reason: $@\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
    exit(1);
}
```

The variables used to define the method’s three arguments were set at previous points in the application file:

The `$config_format` variable was previously set to `xml` unless the `-t` command-line option is included:

```
my $config_format = "xml";
$config_format = "text" if $opt{t};
```

The `$load_action` variable was previously set to `merge` unless the `-a` command-line option is included. The final two lines verify that the specified value is one of the acceptable choices:

```
my $load_action = "merge";
$load_action = $opt{a} if $opt{a};
use constant VALID_ACTIONS => "merge|replace|override";
output_usage() unless ( $load_action =~ /VALID_ACTIONS/);
```

The `$doc` variable was set to the output from the `parsestring` or `parsefile` function (defined in the `XML::DOM::Parser` module), as described in “Read In and Parse the Configuration Data” on page 98.

- The script performs two additional checks for errors and invokes the graceful\_shutdown subroutine in either case:

```

unless ( ref $res ) {
    print "ERROR: Failed to load the configuration from $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

$err = $res->getFirstError();
if ($err) {
    print "ERROR: Failed to load the configuration. Reason: $err->{message}\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}

```

### *Commit the Configuration*

If there are no errors, the script invokes the commit\_configuration method (defined in the file lib/JUNOS/Methods.pm in the JUNOScript Perl distribution):

```

print "Committing configuration from $xmlfile ... \n";
$res = $jnx->commit_configuration();
$err = $res->getFirstError();
if ($err) {
    print "ERROR: Failed to commit configuration. Reason: $err->{message}.\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}

```

## **Parse and Format the Response from the JUNOScript Server**

As the last step in sending a request, the application verifies that there are no errors with the response from the JUNOScript server (see “Submit a Request” on page 94). It can then write the response to a file, to the screen, or both. If the response is for an operational query, the application usually uses XSLT to transform the output into a more readable format, such as HTML or formatted ASCII. If the response consists of configuration data, the application can store it as XML (the JUNOScript tag elements generated by default from the JUNOScript server) or transform it into formatted ASCII.

The following sections discuss parsing and formatting options:

[Parse and Format an Operational Response on page 101](#)

[Parse and Output Configuration Data on page 103](#)

## **Parse and Format an Operational Response**

The following code sample from the `diagnose_bgp.pl` and `get_chassis_inventory.pl` scripts uses XSLT to transform an operational response from the JUNOScript server into a more readable format. A detailed discussion of the functional subsections follows the complete code sample.

```
my $outputfile = $opt{o} || "";
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";

my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);

my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");

if ($nm) {
    print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
    my $command = "xsltproc $nm $deviceinfo{hostname}.xml";

    $command .= "> $outputfile" if $outputfile;
    system($command);
    print "Done\n" if $outputfile;
    print "See $outputfile\n" if $outputfile;
}

else {
    print STDERR "ERROR: Invalid XSL file $xslfile\n";
}
```

The first line of the preceding code sample illustrates how the scripts read the `-o` option from the command line to obtain the name of file into which to write the results of the XSLT transformation:

```
my $outputfile = $opt{o} || "";
```

From the `-x` command-line option, the scripts obtain the name of the XSLT file to use, setting a default value if the option is not provided. The scripts exit if the specified file does not exist. The following example is from the `diagnose_bgp.pl` script:

```
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
```

For examples of XSLT files, see the following directories in the JUNOScript Perl distribution:

The `examples/diagnose_bgp/xsl` directory contains XSLT files for the `diagnose_bgp.pl` script: `dhtml.xsl` generates dynamic HTML, `html.xsl` generates HTML, and `text.xsl` generates ASCII.

The `examples/get_chassis_inventory/xsl` directory contains XSLT files for the `get_chassis_inventory.pl` script: `chassis_inventory_csv.xsl` generates a list of comma-separated values, `chassis_inventory_html.xsl` generates HTML, and `chassis_inventory_xml.xsl` generates XML.

- The actual parsing operation begins by setting the variable \$xmlfile to a filename of the form *router-name.xml* and invoking the printToFile function to write the JUNOScript server's response into the file (the printToFile function is defined in the XML::DOM::Parser module):

```
my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);
```

The next line invokes the translateXSLtoRelease function (defined in the JUNOS::Response module) to alter one of the namespace definitions in the XSLT file. This is necessary because the XSLT 1.0 specification requires that every XSLT file define a specific value for each default namespace used in the data being transformed. The xmlns attribute on a JUNOScript operational response tag element includes a code representing the JUNOS version, such as 5.6R1 for the initial version of JUNOS Release 5.6. Because the same XSLT file can be applied to operational response tag elements from routers running different versions of JUNOS, the XSLT file cannot predefine an xmlns namespace value that matches all versions. The translateXSLtoRelease function alters the namespace definition in the XSLT file identified by the \$xslfile variable to match the value in the JUNOScript server's response. It assigns the resulting XSLT file to the \$nm variable.

```
my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");
```

After verifying that the translateXSLtoRelease function succeeded, the function builds a command string and assigns it to the \$command variable. The first part of the command string invokes the xsltproc command and specifies the names of the XSLT and configuration data files (\$nm and \$deviceinfo{hostname}.xml):

```
if ($nm) {
    print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
    my $command = "xsltproc $nm $deviceinfo{hostname}.xml";
```

If the \$outputfile variable is defined (the file for storing the result of the XSL transformation exists), the script appends a string to the \$command variable to write the results of the xsltproc command to the file. (If the file does not exist, the script writes the results to standard out [stdout].) The script then invokes the system function to execute the command string and prints status messages to stdout.

```
$command .= "> $outputfile" if $outputfile;
system($command);
print "Done\n" if $outputfile;
print "See $outputfile\n" if $outputfile;
}
```

If the translateXSLtoRelease function fails (the if (\$nm) expression evaluates to "false"), the script prints an error:

```
else {
    print STDERR "ERROR: Invalid XSL file $xslfile\n";
}
```

## **Parse and Output Configuration Data**

The get\_config.pl script uses the outconfig subroutine to write the configuration data obtained from the JUNOScript server to a file as either JUNOScript tags or formatted ASCII.

The outconfig subroutine takes four parameters. Three must have defined values: the directory in which to store the output file, the router hostname, and the XML DOM tree (the configuration data) returned by the JUNOScript server. The fourth parameter indicates whether to output the configuration as formatted ASCII, and has a null value if the output should be JUNOScript tag elements. In the following code sample, the script obtains values for the four parameters and passes them to the subroutine. A detailed discussion of each line follows the complete code sample.

```
my(%opt,$login,$password);

getopts('l:p:dm:hit', \%opt) || output_usage();
output_usage() if $opt{h};

my $basepath = shift || output_usage;

my $hostname = shift || output_usage;

my $config = getconfig( $hostname, $jnx, $opt{t} );

outconfig( $basepath, $hostname, $config, $opt{t} );
```

In the first lines of the preceding sample code, the get\_config.pl script uses the following statements to obtain values for the four parameters to the outconfig subroutine:

If the user provides the -t option on the command line, the getopts subroutine records it in the %opt hash. The value keyed to \$opt{t} is passed as the fourth parameter to the outconfig subroutine. (For more information about reading options from the command line, see “Example: Collect Parameters Interactively” on page 86.)

```
getopts('l:p:dm:hit', \%opt) || output_usage();
```

The following line reads the first element of the command line that is not an option preceded by a hyphen. It assigns the value to the \$basepath variable, defining the name of the directory in which to store the file containing the output from the outconfig subroutine. The variable value is passed as the first parameter to the outconfig subroutine.

```
my $basepath = shift || output_usage;
```

The following line reads the next element on the command line. It assigns the value to the \$hostname variable, defining the router hostname. The variable value is passed as the second parameter to the outconfig subroutine.

```
my $hostname = shift || output_usage;
```

The following line invokes the getconfig subroutine to obtain configuration data from the JUNOScript server on the specified router, assigning the resulting XML DOM tree to the \$config variable. The variable value is passed as the third parameter to the outconfig subroutine.

```
my $config = getconfig( $hostname, $jnx, $opt{t} );
```

- The following code sample invokes and defines the outconfig subroutine. A detailed discussion of each functional subsection in the subroutine follows the complete code sample.

```

outconfig( $basepath, $hostname, $config, $opt{t} );

sub outconfig( $$$$ ) {
    my $leader = shift;
    my $hostname = shift;
    my $config = shift;
    my $text_mode = shift;
    my $trailer = "xmlconfig";
    my $filename = $leader . "/" . $hostname . "." . $trailer;

    print "# storing configuration for $hostname as $filename\n";

    my $config_node;
    my $top_tag = "configuration";
    $top_tag .= "-text" if $text_mode;
    if ($config->getTagName() eq $top_tag) {
        $config_node = $config;
    } else {
        print "# unknown response component ", $config->getTagName(), "\n";
    }

    if ( $config_node && $config_node ne "" ) {
        if ( open OUTPUTFILE, ">$filename" )  {
            if (!$text_mode) {
                print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
                print OUTPUTFILE $config_node->toString(), "\n";
            } else {
                my $buf = $config_node->getFirstChild()->toString();
                $buf =~ s/($char_class)/$escapes{$1}/ge;
                print OUTPUTFILE "$buf\n";
            }
            close OUTPUTFILE;
        }
        else {
            print "ERROR: could not open output file $filename\n";
        }
    }
    else {
        print "ERROR: empty configuration data for $hostname\n";
    }
}

```

The first lines of the outconfig subroutine read in the four parameters passed in when the subroutine is invoked, assigning each to a local variable:

```

outconfig( $basepath, $hostname, $config, $opt{t} );

sub outconfig( $$$$ ) {
    my $leader = shift;
    my $hostname = shift;
    my $config = shift;
    my $text_mode = shift;

```

The subroutine constructs the name of the file to which to write the subroutine's output and assigns the name to the \$filename variable. The filename is constructed from the first two parameters (the directory name and hostname) and the \$trailer variable, resulting in a name of the form *directory-name/hostname.xmlconfig*:

```
my $trailer = "xmlconfig";
my $filename = $leader . "/" . $hostname . "." . $trailer;

print "# storing configuration for $hostname as $filename\n";
```

The subroutine checks that the first tag in the XML DOM tree correctly indicates the type of configuration data in the file. If the user included the -t option on the command line, the first tag should be <configuration-text> because the file contains formatted ASCII configuration statements; otherwise, the first tag should be <configuration> because the file contains JUNOScript tag elements. The subroutine sets the \$top\_tag variable to the appropriate value depending on the value of the \$text\_mode variable (which takes its value from opt{t}, passed as the fourth parameter to the subroutine). The subroutine invokes the getTagName function (defined in the XML::DOM::Element module) to retrieve the name of the first tag in the input file, and compares the name to the value of the \$top\_tag variable. If the comparison succeeds, the XML DOM tree is assigned to the \$config\_node variable. Otherwise, the subroutine prints an error message because the XML DOM tree is not valid configuration data.

```
my $config_node;
my $top_tag = "configuration";
$top_tag = "-text" if $text_mode;
if ($config->getTagName() eq $top_tag) {
    $config_node = $config;
} else {
    print "# unknown response component ", $config->getTagName(), "\n";
}
```

The subroutine then uses several nested if statements. The first if statement verifies that the XML DOM tree exists and contains data:

```
if ( $config_node && $config_node ne "" ) {
    ... actions if XML DOM tree contains data ...
}
else {
    print "ERROR: empty configuration data for $hostname\n";
}
```

If the XML DOM tree contains data, the subroutine verifies that the output file can be opened for writing:

```
if ( open OUTPUTFILE, ">$filename" ) {
    ... actions if output file is writable ...
}
else {
    print "ERROR: could not open output file $filename\n";
}
```

If the output file can be opened for writing, the script writes the configuration data into it. If the user requested JUNOScript tag elements (the \$text\_mode variable does not have a value because the user did not include the -t option on the command line), the script writes the string <?xml version=1.0?> as the first line in the output file, then invokes the `toString` function (defined in the XML::DOM module) to write each JUNOScript tag element in the XML DOM tree on a line in the output file:

```
if (!$text_mode) {
    print OUTPUTFILE "<?xml version=\\"1.0\\"?>\n";
    print OUTPUTFILE $config_node->toString(), "\n";
```

If the user requested formatted ASCII, the script invokes the `getFirstChild` and `toString` functions (defined in the XML::DOM module) to write the content of each tag on its own line in the output file. The script substitutes predefined entity references for disallowed characters (which are defined in the %escapes hash), writes the output to the output file, and closes the output file. (For information about defining the %escapes hash to contain the set of disallowed characters, see “Convert Disallowed Characters” on page 89.)

```
} else {
    my $buf = $config_node->getFirstChild()->toString();
    $buf =~ s/($char_class)/$escapes{$1}/ge;
    print OUTPUTFILE "$buf\n";
}
close OUTPUTFILE;
```

## ***Close the Connection to the JUNOScript Server***

To end the JUNOScript session and close the connection to the router, each sample script invokes the `request_end_session` and `disconnect` methods. Several of the scripts do this in standalone statements:

```
$jnx->request_end_session();
$jnx->disconnect();
```

The `load_configuration.pl` script invokes the `graceful_shutdown` subroutine instead (for more information, see “Handle Error Conditions” on page 96):

```
graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_SUCCESS);
```

## **Summary of Mappings Between Perl Queries and JUNOScript Tag Elements**

The sample scripts described in “Overview of the JUNOS Module and Sample Scripts” on page 81 invoke only a small number of the predefined JUNOScript Perl queries that client applications can use. Table 6 maps all of the Perl queries available in the current version of JUNOScript to the corresponding JUNOScript response tag element and JUNOS CLI command. Each query has the same name as the corresponding JUNOScript request tag element (to derive the name of the request tag element, replace each underscore with a hyphen and enclose the string in angle brackets).

For more information about JUNOScript request and response tag elements, see the *JUNOScript API Reference*.

**Table 6: Mapping of Perl/Java Methods to JUNOScript Tags**

Method	Response Tag	CLI Command
clear_arp_table	<clear-arp-table-results>	clear arp
clear_helper_statistics_information	NONE	clear helper statistics
clear_ipv6_nd_information	<ipv6-modify-nd>	clear ipv6 neighbors
file_compare	NONE	file compare
file_copy	NONE	file copy
file_delete	NONE	file delete
file_list	<directory-list>	file list
file_rename	NONE	file rename
file_show	<file-content>	file show
get_accounting_profile_information	<accounting-profile-information>	show accounting profile
get_accounting_record_information	<accounting-record-information>	show accounting records
get_alarm_information	<alarm-information>	show chassis alarms
get_arp_table_information	<arp-table-information>	show arp
get_bgp_group_information	<bgp-group-information>	show bgp group
get_bgp_neighbor_information	<bgp-information>	show bgp neighbor
get_bgp_summary_information	<bgp-information>	show bgp summary
get_chassis_inventory	<chassis-inventory>	show chassis hardware
get_cos_classifier_information	<cos-classifier-information>	show class-of-service classifier
get_cos_classifier_table_information	<cos-classifier-table-information>	show class-of-service forwarding-table classifier
get_cos_classifier_table_map_information	<cos-classifier-table-map-information>	show class-of-service forwarding-table classifier mapping
get_cos_code_point_map_information	<cos-code-point-map-information>	show class-of-service code-point-aliases
get_cos_drop_profile_information	<cos-drop-profile-information>	show class-of-service drop-profile
get_cos_fabric_scheduler_map_information	<cos-fabric-scheduler-map-information>	show class-of-service fabric scheduler-map
get_cos_forwarding_class_information	<cos-forwarding-class-information>	show class-of-service forwarding-class
get_cos_fwtab_fabric_scheduler_map_information	<cos-fwtab-fabric-scheduler-map-information>	show class-of-service forwarding-table fabric scheduler-map
get_cos_information	<cos-information>	show class-of-service
get_cos_interface_map_information	<cos-interface-information>	show class-of-service interface
get_cos_red_information	<cos-red-information>	show class-of-service forwarding-table drop-profile
get_cos_rewrite_information	<cos-rewrite-information>	show class-of-service rewrite-rule
get_cos_rewrite_table_information	<cos-rewrite-table-information>	show class-of-service forwarding-table rewrite-rule
get_cos_rewrite_table_map_information	<cos-rewrite-table-map-information>	show class-of-service forwarding-table rewrite-rule mapping
get_cos_scheduler_map_information	<cos-scheduler-map-information>	show class-of-service scheduler-map
get_cos_scheduler_map_table_information	<cos-scheduler-map-table-information>	show class-of-service forwarding-table scheduler-map
get_cos_table_information	<cos-table-information>	show class-of-service forwarding-table
get_destination_class_statistics	<destination-class-statistics>	show interfaces destination-class
get_environment_information	<environment-information>	show chassis environment
get_fabric_queue_information	<fabric-queue-information>	show class-of-service fabric statistics
get_feb_information	<scb-information>	show chassis feb

Method	Response Tag	CLI Command
get_firewall_information	<firewall-information>	show firewall
get_firewall_log_information	<firewall-log-information>	show firewall log
get_firewall_prefix_action_information	<firewall-prefix-action-information>	show firewall prefix-action-stats
get_firmware_information	<firmware-information>	show chassis firmware
get_forwarding_table_information	<forwarding-table-information>	show route forwarding-table
get_fpc_information	<fpc-information>	show chassis fpc
get_ggsn_apn_statistics_information	<apn-statistics-information>	show services ggsn statistics apn
get_ggsn_gtp_prime_statistics_information	<gtp-prime-statistics-information>	show services ggsn statistics gtp-prime
get_ggsn_gtp_statistics_information	<gtp-statistics-information>	show services ggsn statistics gtp
get_ggsn_imsi_trace	<call-trace-information>	show services ggsn trace imsi
get_ggsn_imsi_user_information	<mobile-user-information>	show services ggsn statistics imsi
get_ggsn_interface_information	<ggsn-interface-information>	show services ggsn status
get_ggsn_msisdn_trace	<call-trace-information>	show services ggsn trace msisdn
get_ggsn_sgsn_statistics_information	<sgsn-statistics-information>	show services ggsn statistics sgsn
get_ggsn_statistics	<ggsn-statistics>	show services ggsn statistics
get_ggsn_trace	<call-trace-information>	show services ggsn trace all
get_helper_statistics_information	<helper-statistics-information>	show helper statistics
get_ike_security_associations_information	<ike-security-associations-information>	show ike security-associations
get_instance_information	<instance-information>	show route instance
get_interface_filter_information	<interface-filter-information>	show interfaces filters
get_interface_information	<interface-information>	show interfaces
get_interface_policer_information	<interface-policer-information>	show interfaces policers
get_interface_queue_information	<interface-information>	show interfaces queue
get_ipv6_nd_information	<ipv6-nd-information>	show ipv6 neighbors
get_ipv6_ra_information	<ipv6-ra-information>	show ipv6 router-advertisement
get_isis_adjacency_information	<isis-adjacency-information>	show isis adjacency
get_isis_database_information	<isis-database-information>	show isis database
get_isis_hostname_information	<isis-hostname-information>	show isis hostname
get_isis_interface_information	<isis-interface-information>	show isis interface
get_isis_route_information	<isis-route-information>	show isis route
get_isis_spf_information	<isis-spf-information>	show isis spf
get_isis_statistics_information	<isis-statistics-information>	show isis statistics
get_l2ckt_connection_information	<l2ckt-connection-information>	show l2circut connections
get_l2vpn_connection_information	<l2vpn-connection-information>	show l2vpn connections
get_ldp_database_information	<ldp-database-information>	show ldp database
get_ldp_interface_information	<ldp-interface-information>	show ldp interface
get_ldp_neighbor_information	<ldp-neighbor-information>	show ldp neighbor
get_ldp_path_information	<ldp-path-information>	show ldp path
get_ldp_route_information	<ldp-route-information>	show ldp route
get_ldp_session_information	<ldp-session-information>	show ldp session
get_ldp_statistics_information	<ldp-statistics-information>	show ldp statistics
get_ldp_traffic_statistics_information	<ldp-traffic-statistics-information>	show ldp traffic-statistics
get_lm_information	<lm-information>	show link-management

Method	Response Tag	CLI Command
get_lm_peer_information	<lm-peer-information>	show link-management peer
get_lm_routing_information	<lm-information>	show link-management routing
get_lm_routing_peer_information	<lm-peer-information>	show link-management routing peer
get_lm_routing_te_link_information	<lm-te-link-information>	show link-management routing te-link
get_lm_te_link_information	<lm-te-link-information>	show link-management te-link
get_mpls_admin_group_information	<mpls-admin-group-information>	show mpls admin-groups
get_mpls_cspf_information	<mpls-cspf-information>	show mpls cspf
get_mpls_interface_information	<mpls-interface-information>	show mpls interface
get_mpls_lsp_information	<mpls-lsp-information>	show mpls lsp
get_mpls_path_information	<mpls-path-information>	show mpls path
get_ospf_database_information	<ospf-database-information>	show ospf database
get_ospf_interface_information	<ospf-interface-information>	show ospf interface
get_ospf_io_statistics_information	<ospf-io-statistics-information>	show ospf io-statistics
get_ospf_log_information	<ospf-log-information>	show ospf log
get_ospf_neighbor_information	<ospf-neighbor-information>	show ospf neighbor
get_ospf_route_information	<ospf-route-information>	show ospf route
get_ospf_statistics_information	<ospf-statistics-information>	show ospf statistics
get_ospf3_database_information	<ospf3-database-information>	show ospf3 database
get_ospf3_interface_information	<ospf3-interface-information>	show ospf3 interface
get_ospf3_io_statistics_information	<ospf3-io-statistics-information>	show ospf3 io-statistics
get_ospf3_log_information	<ospf3-log-information>	show ospf3 log
get_ospf3_neighbor_information	<ospf3-neighbor-information>	show ospf3 neighbor
get_ospf3_route_information	<ospf3-route-information>	show ospf3 route
get_ospf3_statistics_information	<ospf3-statistics-information>	show ospf3 statistics
get_passive_monitoring_error_information	<passive-monitoring-error-information>	show passive-monitoring error
get_passive_monitoring_flow_information	<passive-monitoring-flow-information>	show passive-monitoring flow
get_passive_monitoring_information	<passive-monitoring-information>	show passive-monitoring
get_passive_monitoring_memory_information	<passive-monitoring-memory-information>	show passive-monitoring memory
get_passive_monitoring_status_information	<passive-monitoring-status-information>	show passive-monitoring status
get_passive_monitoring_usage_information	<passive-monitoring-usage-information>	show passive-monitoring usage
get_pic_detail	<pic-information>	show chassis pic
get_pic_information	<fpc-information>	show chassis fpc pic-status
get_rmon_alarm_information	<rmon-alarm-information>	show snmp rmon alarms
get_rmon_event_information	<rmon-event-information>	show snmp rmon events
get_rmon_information	<rmon-information>	show snmp rmon
get_route_engine_information	<route-engine-information>	show chassis routing-engine
get_route_information	<route-information>	show route
get_route_summary_information	<route-summary-information>	show route summary
get_rsvp_interface_information	<rsvp-interface-information>	show rsvp interface
get_rsvp_neighbor_information	<rsvp-neighbor-information>	show rsvp neighbor
get_rsvp_session_information	<rsvp-session-information>	show rsvp session
get_rsvp_statistics_information	<rsvp-statistics-information>	show rsvp statistics
get_rsvp_version_information	<rsvp-version-information>	show rsvp version

Method	Response Tag	CLI Command
get_rtexport_instance_information	<rtexport-instance-information>	show route export instance
get_rtexport_table_information	<rtexport-table-information>	show route export
get_rtexport_target_information	<rtexport-target-information>	show route export vrf-target
get_scb_information	<scb-information>	show chassis scb
get_security_associations_information	<security-associations-information>	show ipsec security-associations
get_sfsm_information	<scb-information>	show chassis sfm
get_snmp_information	<snmp-statistics>	show snmp statistics
get_software_information	<software-information>	show version
get_source_class_statistics	<source-class-statistics>	show interfaces source-class
get_spmb_information	<spmb-information>	show chassis spmb
get_spmb_sib_information	<spmb-sib-information>	show chassis spmb sibs
get_ssbb_information	<scb-information>	show chassis ssb
get_syslog_tag_information	<syslog-tag-information>	help syslog
get_system_storage	<system-storage-information>	show system storage
get_system_uptime_information	<system-uptime-information>	show system uptime
get_system_users_information	<system-users-information>	show system users
get_ted_database_information	<ted-database-information>	show ted database
get_ted_link_information	<ted-link-information>	show ted link
get_ted_protocol_information	<ted-protocol-information>	show ted protocol
request_end_session	<end-session>	quit
request_ggsn_restart_interface	<interface-action-results>	request services ggsn restart interface
request_ggsn_restart_node	<node-action-results>	request services ggsn restart node
request_ggsn_start_imsi_trace	NONE	request services ggsn trace start imsi
request_ggsn_start_msisdn_trace	NONE	request services ggsn trace start msisdn
request_ggsn_stop_imsi_trace	NONE	request services ggsn trace stop imsi
request_ggsn_stop_msisdn_trace	NONE	request services ggsn trace stop msisdn
request_ggsn_stop_trace_activity	NONE	request services ggsn trace stop all
request_ggsn_terminate_context	<pdp-context-deletion-results>	request services ggsn pdp terminate context
request_ggsn_terminate_contexts_apn	<apn-pdp-context-deletion-results>	request services ggsn pdp terminate apn
request_halt	NONE	request system halt
request_package_add	NONE	request system software add
request_package_delete	NONE	request system software delete
request_package_validate	NONE	request system software validate
request_reboot	NONE	request system reboot
request_snapshot	NONE	request system snapshot